Simulation Methods - Numerical methods for ODE

Francesc Parcerisas Vela

April 8, 2025

1 Introduction

In this document, we describe the implementation of two numerical integration methods used to solve a Cauchy problem from a ordinary differential equation. The two methods are:

- Runge-Kutta-Fehlberg of orders 4 and 5
- Taylor-method of order $n \in \mathbb{N}$

The example equation we will be testing the method on is the following differential equation defined over $\mathbb{R}^2$

$$\dot{x} = y, \qquad \dot{y} = -ky - x^3 + b_0 + b_1 \cos(t), \tag{1}$$

where $k = 0.08, b_0 = 4, b_1 = 15.77$

2 RKF4(5)

2.1 Algorithm Overview

Given an initial time $t_j \in \mathbb{R}$ and point $x_j \in \mathbb{R}^2$ and initial increment h_j, the method follows these steps:

1. The method starts by computing the six intermediate values k_i :

$$
\begin{aligned}
k_1 &= f(t_j, x_j),\\
k_2 &= f\left(t_j + \frac{1}{4}h_j,\, x_j + h_j\left(\frac{1}{4}k_1\right)\right),\\
k_3 &= f\left(t_j + \frac{3}{8}h_j,\, x_j + h_j\left(\frac{3}{32}k_1 + \frac{9}{32}k_2\right)\right),\\
k_4 &= f\left(t_j + \frac{12}{13}h_j,\, x_j + h_j\left(\frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3\right)\right),\\
k_5 &= f\left(t_j + h_j,\, x_j + h_j\left(\frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4\right)\right),\\
k_6 &= f\left(t_j + \frac{1}{2}h_j,\, x_j + h_j\left(-\frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5\right)\right).
\end{aligned}
$$

2. Then it computes $\hat{x}_{j+1}$, $\tilde{x}_{j+1}$ as

$$
\begin{aligned}
\tilde{x}_{j+1} &= \tilde{x}_j + h_j\left(\frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5\right),\\
\hat{x}_{j+1} &= \tilde{x}_j + h_j\left(\frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6\right).
\end{aligned}
$$

3. Given an error bound ϵ_{j+1}, then:

   - If $\|\hat{x}_{j+1} - \tilde{x}_{j+1}\|_2 \leq \epsilon_{j+1}$, take $h_{j+1} = h_N$ and proceed.
   - Else, take $h_j = h_N$ and repeat step 1.

   where $h_N = 0.9 \cdot h_j \cdot \sqrt[5]{\dfrac{\epsilon_{j+1}}{\|\hat{x}_{j+1} - \tilde{x}_{j+1}\|_2}}$.

Following these steps, the method allows to provide an estimate of function $x(t)$ at each point t_j as $x(t_j) \approx x_j$.

2.2 Code implementation

In the implementation we follow a straightforward structure. Iteratively, we compute the k_i from the definition:

```
// Compute k1
ode(t, x, n, k[0]);

// For each stage j
for (int j = 1; j < STAGES; j++) {
    for (int i = 0; i < n; i++) {
        xtemp[i] = x[i];
        for (int l = 0; l < j; l++) {
```

```
            xtemp[i] += h * a[j][l] * k[l][i];
        }
    }
    ode(t + c[j]*h, xtemp, n, k[j]);
}
```

Then, we compute $\hat{x}_{n+1}$, $\tilde{x}_{n+1}$ and estimate the error, again using the given formulas:

```
//Calculate x4
for (int i = 0; i < n; i++) {
            double b_ = 0.0;
    for (int j = 0; j < STAGES; j++) {
        b_ += b_4[j] * k[j][i];
    }
    x4[i] = x[i] + h * b_;
}
//Calculate x5
for (int i = 0; i < n; i++) {
    double b_ = 0.0;
    for (int j = 0; j < STAGES; j++) {
        b_ += b_5[j] * k[j][i];
    }
    x5[i] = x[i] + h * b_;
}

//Compute estimated error
double err = 0.0;
for (int i = 0; i<n; i++){
    err_vec[i] = (1.0/360.0) * k[0][i] - (128.0/4275.0) * k[2][i] -
       (2197.0/75240.0) * k[3][i]
                + (1.0/50.0) * k[4][i] + (2.0/55.0) * k[5][i];
    err += err_vec[i] * err_vec[i];
}
err = sqrt(err) * fabs(h); // estimated error
```

Finally, if the error is small enough, the position (x, y) is updated. Otherwise, the step is recalculated and the algorithm starts all over with the new step.

```
double hN = 0.9 * h * pow(tol / err, 0.2); // new stepsize
if(!sc ||(sc && err < tol)){ // stepsize control
   step_accepted = 1; // step accepted
   t += h; // update time
    for (int i = 0; i < n; i++) {
         x[i] = x4[i]; // update position
    }
} else { // error too large, reduce step size
    h = hN; // reduce step size
    if (atf && t + h > *atf) { // check if new step size is too large
        h = *atf - t; // reduce step size to reach atf
        result = 1; // indicate that we reached atf
    }
}
```

2.3 Results

After running RKF4(5) method with equation (1), starting point $(0, 0)$ and up to $t = 20$, we obtain the plot on Figure 1.

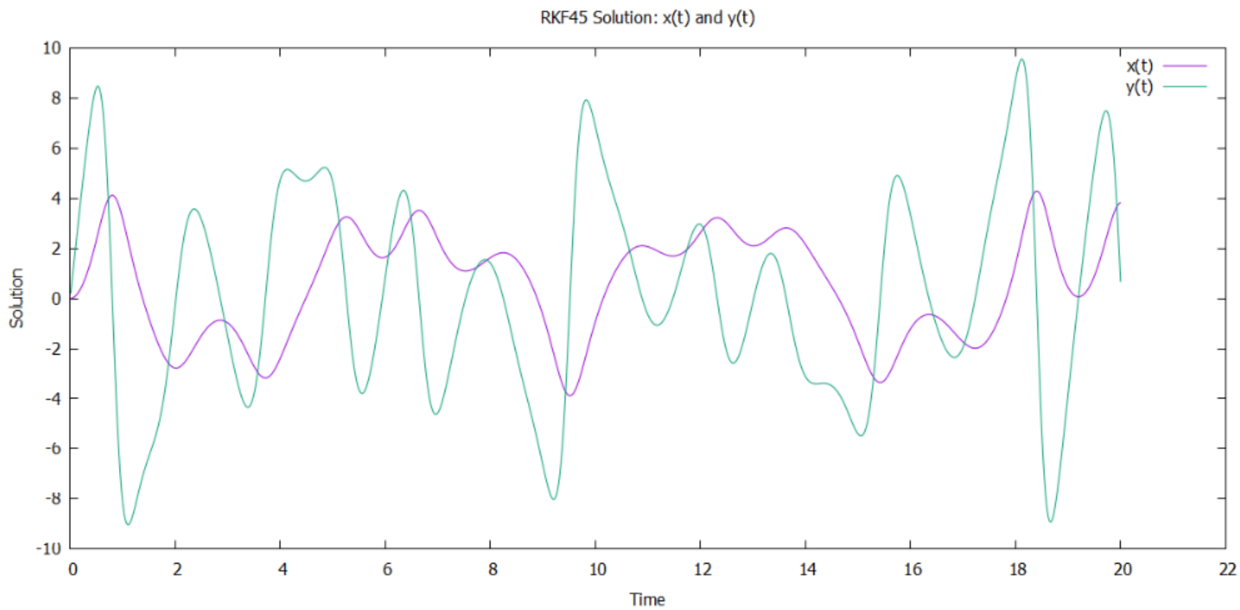


Figure 1: Solution to IVP (1) with $(x_0, y_0) = (0, 0)$, $t_0 = 0$

Visually, we can tell that the solution might be right since the extreme points of $x(t)$ are the same points where $y(t) = 0$. Hence, it is consistent with the fact that $x'(t) = y(t)$.

3 Taylor's method

3.1 Algorithm overview

Given the nature of the ode (1), we can easily compute some of the derivatives of x with respect to t. That is:

$$\begin{aligned}
x'(t) &= y(t), \\
x''(t) &= -ky(t) - x^3(t) + b_0 + b_1 \cos(t), \\
x'''(t) &= -ky'(t) - 3x^2(t)y(t) - b_1 \sin(t), \\
x^{(4)}(t) &= -ky''(t) - 3(2x(t)y^2(t) + x^2(t)y''(t)) - b_1 \cos(t) \\
&\vdots
\end{aligned}$$

Using this, we can take an approximation on each point $x(t + h)$ using Taylor's polynomial of order 4

$$x(t + h) \approx x(t) + hx'(t) + \frac{h^2}{2!}x''(t) + \cdots + \frac{h^n}{n!}x^{(n)}(t)$$

3.2 Code implementation

Code implementation of this method is a straightforward evaluation at each point using the above formula:

```
void taylor_step(double *t, double *x, double *y, double h, int order,
                 void (*taylor_coeffs)(double, double, double, double*,
                    int)) {
    double coeffs[order];
    taylor_coeffs(*t, *x, *y, coeffs, order);

    double h_pow = h;
    for (int i = 0; i < order; i++) {
        x_new += (h_pow / tgamma(i + 2)) * coeffs[i];
        if (i < order - 1) {
            y_new += (h_pow / tgamma(i + 2)) * coeffs[i + 1];
        }
        h_pow *= h;
    }
}
```

For the particular case of the ode (1), the taylor coefficients are computed as follows:

```
void taylor_coeffs(double t, double x, double y, double *coeffs, int order
   ) { //taylor coefficients for example ODE
if (order >= 1)
coeffs[0] = y;

if (order >= 2)
    coeffs[1] = -k * y + b0 - x * x * x + b1 * cos(t);

if (order >= 3)
    coeffs[2] = -k * coeffs[1] - 3 * x * x * y - b1 * sin(t);

if (order >= 4)
    coeffs[3] = -k * coeffs[2] - 3 * (2 * x * y * y + x * x * coeffs[1]) -
        b1 * cos(t);
}
```

3.3 Results

After running Taylor's method with equation (1), starting point $(0, 0)$ and up to $t = 20$, we obtain the plot on Figure (2).
This plot is consistent with the one obtained with RKF45 which might indicate we got a good solution.

4 Poincaré maps

Let ϕ be the flow associated to a ode. Then, we compute the maps $P(x, y) = \phi(x, y, 2\pi)$. Using the implementation of Taylor method, we implement a way to compute the maps and

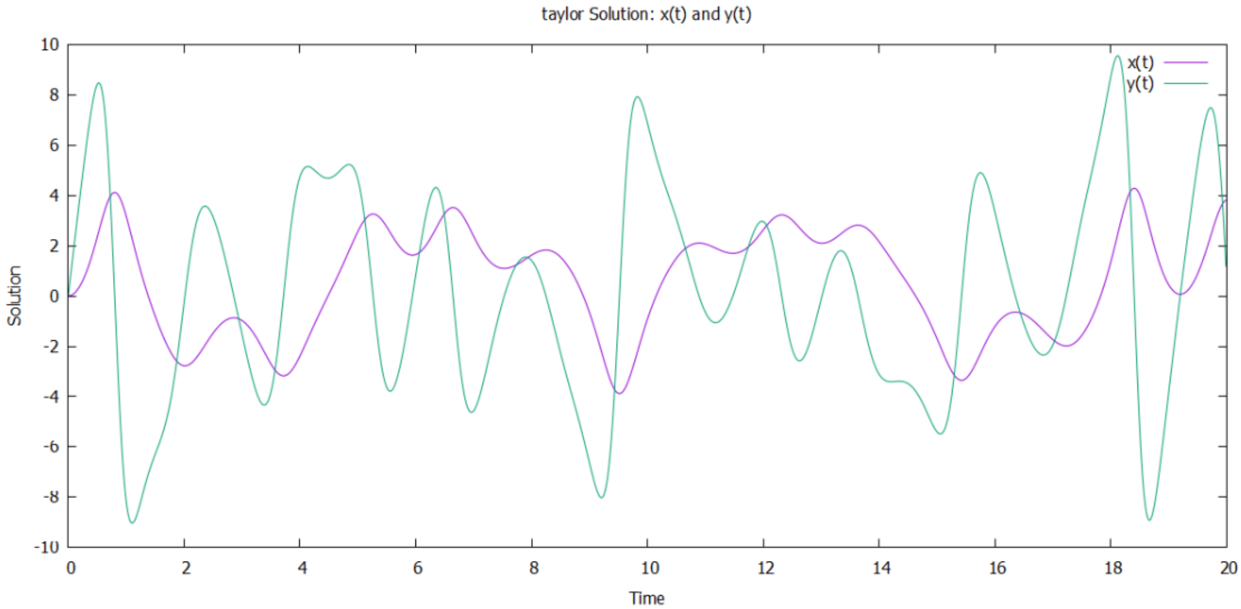


Figure 2: Solution to IVP (1) with $(x_0, y_0) = (0, 0)$, $t_0 = 0$

store them in a `.dat` file:

```
while (*t < tf) { // Taylor method loop
    fprintf(gnuplot_file, "%.10f\t%.10f\n", x[0], x[1]); //write results
    taylor_step(t, x, y, h, order, taylor_coeffs); // perform Taylor step
}
```

Considering the case of the ode (1), we compute this map from time t_0 up to time $t = 2\pi$ and initial condition $(x_0, y_0) = (0, 0)$. The plot is the one on Figure (3).
Finally in Figure 4 we plot 10000 iterates of maps for different starting in different initial points

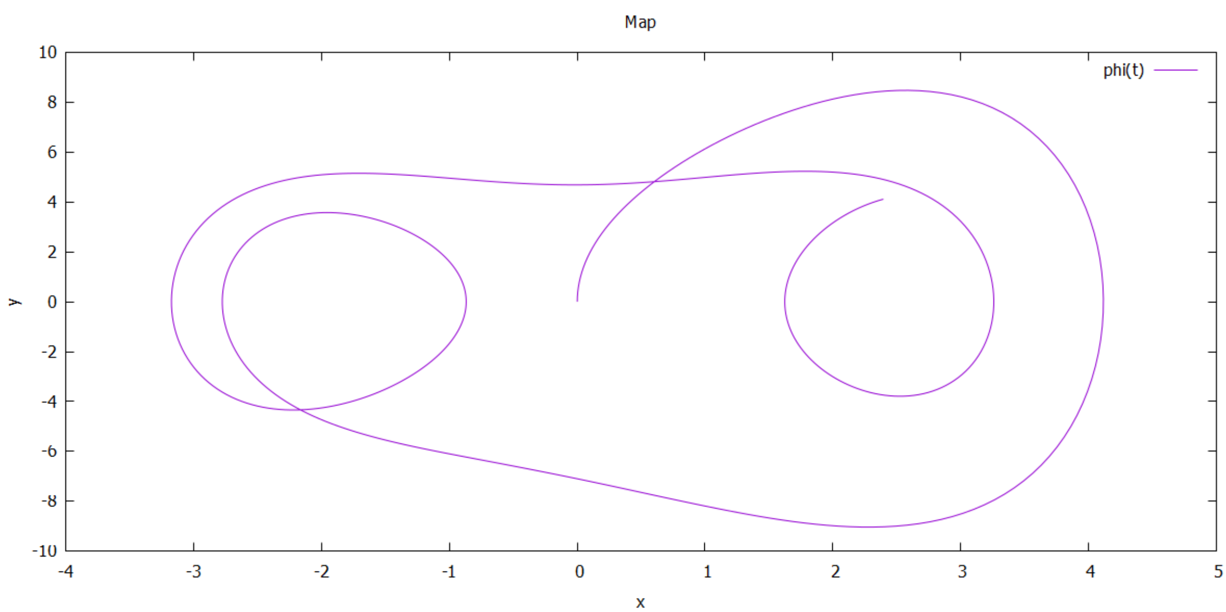
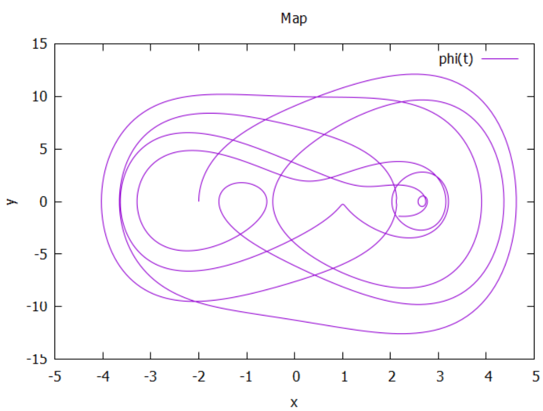
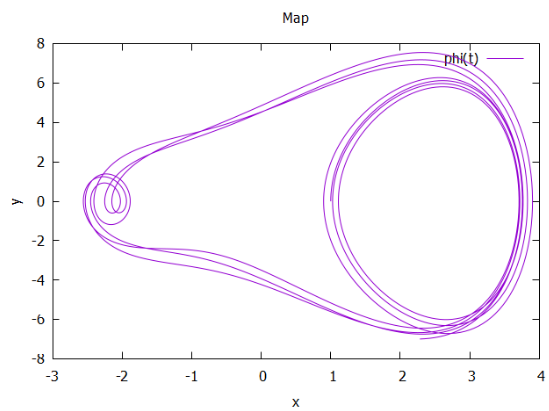


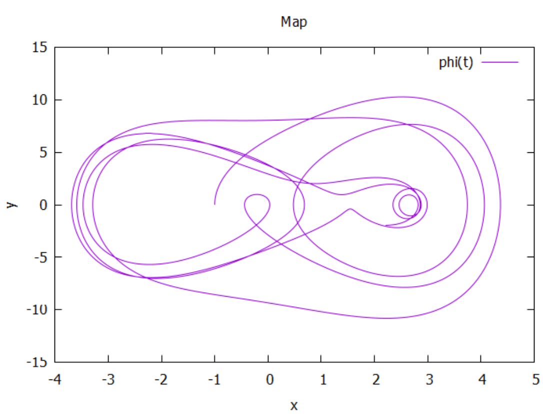
Figure 3: Map of ODE (1) with $(x_0, y_0) = (0, 0)$, $t_0 = 0$, $t = 2\pi$



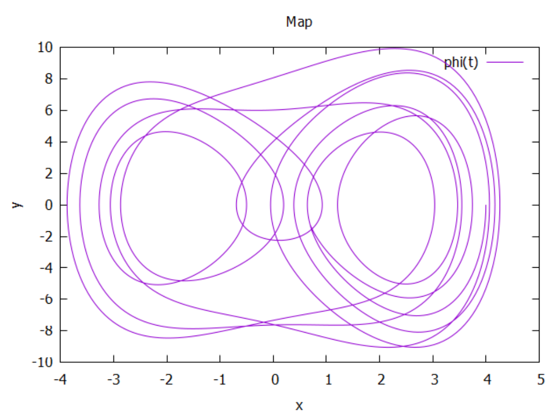
(a) $(x_0, y_0) = (-2, 0)$



(b) $(x_0, y_0) = (-2, 0)$



(c) $(x_0, y_0) = (-1, 0)$



(d) $(x_0, y_0) = (4, 0)$

Figure 4: Maps of (1) for different initial conditions